AFRL-IF-RS-TR-2006-140
Final Technical Report
April 2006

QUICKSILVER: MIDDLEWARE FOR SCALABLE SELF-REGENERATIVE SYSTEMS

Cornell University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. S467

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2006-140 has been reviewed and is approved for publication.

APPROVED: /s/

PATRICK M. HURLEY
Project Engineer

FOR THE DIRECTOR: /s/

WARREN H. DEBANY, Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	APRIL 2006	Final Jul 04 – Jan 06

4. TITLE AND SUBTITLE
QUICKSILVER: MIDDLEWARE FOR SCALABLE SELF-REGENERATIVE SYSTEMS

5. FUNDING NUMBERS
C - FA8750-04-2-0011
PE - 62301E
PR - S467
TA - SR
WU - SP

6. AUTHOR(S)
Kenneth P. Birman

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Cornell University
120 Day Hall
Ithaca New York 14853

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington Virginia 22203-1714

AFRL/IFGA
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2006-140

11. SUPPLEMENTARY NOTES
AFRL Project Engineer: Patrick M. Hurley/IFGA/(315) 330-3624/ Patrick.Hurley@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

Our project was motivated by scalability and robustness gaps identified in the GIG/NCES COTS technology base. Concerns included the instability and poor performance of publish-subscribe technology in large deployments and the challenges of building scalable cluster-hosted technologies for data centers under intensive time-critical stress. Our solutions include the Ricochet real-time multicast, Cayuga message filter, a new scalable services architecture, the Quicksilver scalable multicast, Chunkyspread and Fireflies. Several of these are now being distributed for use by our partners in the military and elsewhere and will also be used by Cornell for follow-on projects. In addition to software solutions, our group participated in government workshops and worked closely with the Air Force on a transitioning path to move our solutions into their hands, and is likely to receive some continuing funding to pursue that path. Moreover, we identified promising applications for Ricochet in the context of the Navy DD (X) program. Finally, we explored some tangential applications of SRS concepts in the context of mobility. This ultimately led to a successful proposal under the DARPA ACERT program, managed by Dr. Jonathan Smith (IFTO).

14. SUBJECT TERMS
Scalable GIG/NCES platform technologies, time-critical multicast

15. NUMBER OF PAGES
37

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Figures

1. Project Overview

Cornell's Quicksilver effort, a partnership between three academic research teams at the university and a military vendor (Raytheon) funded under a "sibling" contract to this one, responds to trends poised to dramatically reshape the landscape for military computing and communication systems. The Global Information Grid (GIG) and the associated Network Centric Enterprise Systems (NCES) rollout, now gaining momentum, are breaking down communication boundaries between existing stovepipe systems. Service Oriented Architecture (SOA) standards will let us create new kinds of applications by integrating information and capabilities in legacy stovepipe platforms and systems. Against this backdrop one finds a mixture of risks and tremendous opportunity. Our effort seeks to mitigate the risks and open the door to exploiting some opportunities that might otherwise fall flat.

The background for the GIG/NCES developments parallel broader industry trends. A vast array of commercial applications is starting to adopt SOA designs in support of new kinds of integrated distributed systems. By exploiting this broader commercial backdrop, the military has an opportunity to gain efficiencies without developing a non-standard, and hence more costly, technology base.

The recent developments in this area have potential to support broader military priorities. Secretary Rumsfeld has pressed for force modernization and argued that the resulting improved effectiveness, rapid response capabilities and cost savings will transform the military. Within the branches of the military, computing systems are seen as the low hanging fruit and hence the best path to realize this ambitious vision. For example, General Hobbins, at the time CIO of the Air Force, recently wrote that "great networks and great applications" will be vital to Air Force supremacy in the coming decade. As engineers and researchers, our job is to make sure that the networks actually are "great"!

Unfortunately, industry won't necessarily deliver the needed technology on its own, primarily because industry is motivated by economic considerations remote from those that drive the military. Thus, we need to learn to build demanding military applications using the platforms and technology components that will be available in the early GIG/NCES environments, or that are in the pipeline. To some extent, this just comes down to finding non-standard ways to use the tools being developed today by major vendors, such as IBM, Sun and Microsoft. But we also need to develop additional technologies compatible with those tools, in order to extend aspects that are deficient today and likely to remain inadequate long into the future. In particular, military applications will demand exceptional scalability and exceptional real-time responsiveness. These sorts of requirements are uncommon in commercial systems and hence have received relatively little attention from vendors, as documented in [Bir05a, Bir05b, BHP05, vRB05].

Service oriented systems are best understood in terms of a layered architecture. Client computing systems such as PCs operating in vehicles or carried by the individual warrior

run a commodity operating system such as Windows and interact with one-another in either of two primary ways.

- Through the intermediary of a server platform operating in a data center. In this case client systems might interact with the server in a request-response manner, or they might form a longer binding to it and receive notifications and data through streaming media feeds or event upcalls delivered using a publish-subscribe technology.

- Through direct collaborative communication and collaborative action, in which is exchanged directly from client to client over a publish-subscribe message bus.

We can easily identify parallels to this vision in the GIG/NCES architecture. GIG platforms such as the Air Force Joint Battlespace Infosphere (JBI) aim at support for direct client-to-client communication through a secured publish-subscribe capability. For example, the JBI might link a radar system capturing incoming threats to a weapons targeting platform designed to prioritize and neutralize those threats. The NCES side of the architecture corresponds to the server platform part of the SOA approach, and defines the way that future databases or other services would be accessed. For example, a soldier downloading a map of a city where operations will be undertaken would probably do so using a technology very much like the one used when we download documents from web sites using browsers. And indeed, the web services standards that have been selected for the GIG/NCES environment closely parallel web-browser technologies, with the exception that web services also permits computer programs to act as "clients", playing the web browser role, and allows pretty much any service to offer a web services interface at the touch of a button. The term used above, "publish-subscribe", is a catch-all for a class of technologies permitting the application to generate messages, tag them with a topic or other meta-data, and then "publish" them. Applications "subscribe" to specific topics or to a selection filter expressed over meta-data or even the data in the messages themselves (so-called "content filtering"). When a publication matches a subscription, the corresponding client or clients will receive a copy of the message. This paradigm is popular because it is easy to use and because it supports a style of programming in which clients can be implemented independent of servers and then coupled later by the publish-subscribe communication bus.

To scale these technologies in large settings, we need two kinds of solutions. For the server scenarios, we need a way to build a very large data center in which the service used by clients is actually implemented by a family of cooperating server programs scaled over the nodes on the cluster as seen in Figure 1. For publish-subscribe communication directly between client nodes without the intermediary of a server, in contrast, the requirement is simply for a communications architecture that can operate rapidly and reliably with large numbers of users and under the potential stresses seen during battle, such as damage to the communication nodes, overloads, etc.

Notice that the figure makes two uses of publish-subscribe. This is paralleled in our own work, which has yielded (among other technologies), two communications platforms.

One (we call it Ricochet) is in support of uses *within* a data center or cluster, where the emphasis is on real-time computing and communication. The other is for communication from servers to clients.

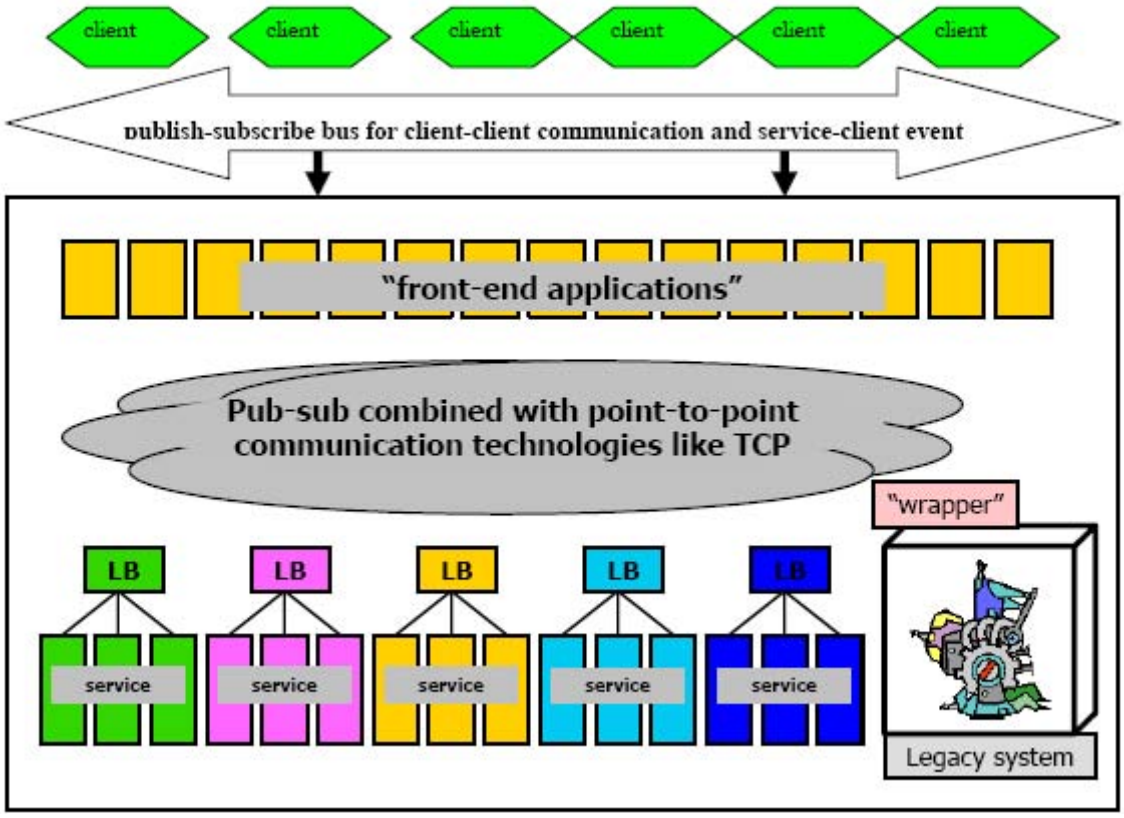

Figure 1: Scalable clustered server platform.

A goal of the Cornell effort is to simplify the development of scalable clustered server platforms such as the one shown here (Figure 1). The server is physically operated as a data center and includes racks of computing nodes – more can be added as needed. Applications need to "scale" as transparently as possible, permitting queries to be load-balanced over the backend compute nodes; a front-end service builds the web pages seen by users or routes web-service requests to the appropriate back-end services. Legacy applications are supported by wrapping them in a software layer that can interact with the surrounding web-services framework, and publish-subscribe is used throughout. Clients can communicate directly over a publish-subscribe message bus, or can interact with servers running in data centers using web services standards. Those servers support both a request-reply style of computing and various forms of streaming, including use of the message bus to report events back to the clients. Internally to the data center one sees a second kind of publish-subscribe platform, used between programs running within the center. The properties expected of these various communication systems vary to match the specific needs and guarantees of the applications, the threat models they need to tolerate, and the style of management/administration appropriate to the setting.

Our premise is that current off-the-shelf technologies have made it much too hard to implement these kinds of systems, and that the technologies themselves often scale poorly. Cornell's contributions are at several levels of this picture: faster content filtering mechanisms for use in the publish-subscribe layer (Cayuga), a new way to built time-critical web services (Ricochet and Tempest), a new and more scalable publish-subscribe technology base (Quicksilver), and new ways to build overlay networks for purposes such as system monitoring (Fireflies) or media streaming (ChunkySpread). Of these, Cayuga, Ricochet and Tempest target the interior of a data center like the one shown. Quicksilver, Fireflies and ChunkySpread focus on direct client-to-client data paths and communication between a server and its widely distributed client computing systems. The DARPA funding needed to integrate these components into a single system was part of an unfunded optional extension, but our project did produce the many component technologies just cited and we have a growing user community both within the military (notably the Air Force and, through Raytheon, the Navy DD(X) program) and in vendor settings.

2. Scalable Communication Platforms: Ricochet and Quicksilver

Our work on the Quicksilver and Ricochet communication frameworks and on Tempest, a new system we're currently building over Ricochet, respond to the GIG and NCES trends just summarized. Ricochet and Quicksilver introduce quality of service and scalability guarantees into GIG/NCES servers and event streams, with the goal of increasing predictability of these systems, reducing response time for time-critical applications, and simplifying the life of the developer charged with implementing a new service.

- *Ricochet* is a multicast layer in support of event notification with time-critical communication requirements. A primary use will be in support of Tempest, a new system for automating the development of scalable time-critical clustered services (Figure 1, 2).

- *Quicksilver* is a scalable communication system for WAN or LAN settings (Figure 3). It supports a publish-subscribe style of communication in which applications running directly on PCs in a WAN can stream data at high rates to one-another. The focus here is on raw throughput and stability during disruptive episodes, but not on time-critical event delivery.

Both systems are currently libraries of procedures to which applications can be linked directly. We are about to do a general release of Ricochet, with the team doing DD(X) eager to pick up and experiment with the technology for time-critical notification purposes on that platform, and several other potential users expressing interest at Yahoo!, Sun Microsystems and Amazon. We are also using Ricochet in support of *Tempest*, a new drag-and-drop technology for programming clustered computers that combines Ricochet with a new scalable services architecture described in [MBvR05].

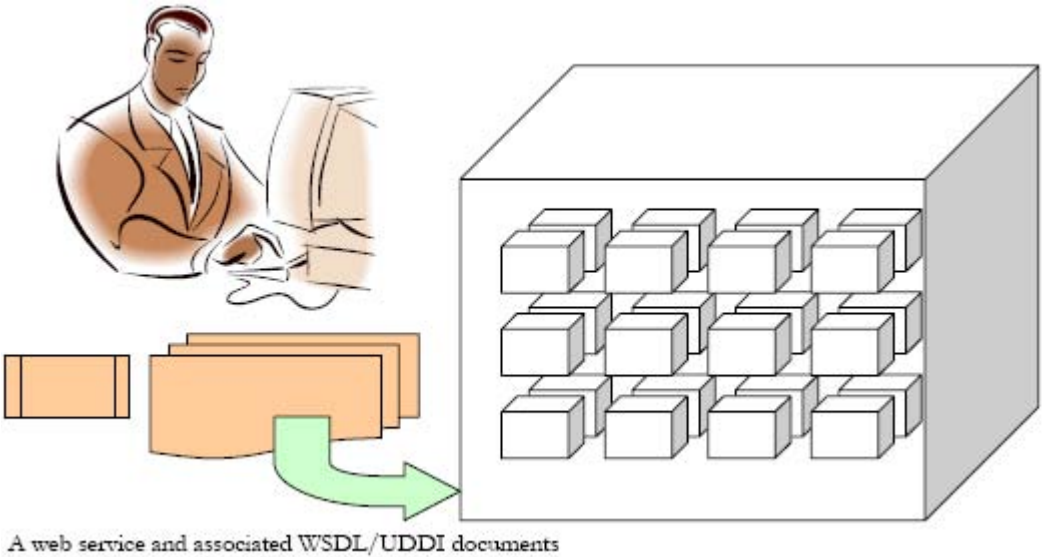

Figure 2: Tempest enables drag and drop development for building scalable, time-critical web services that will run on clusters in GIG/NCES settings.

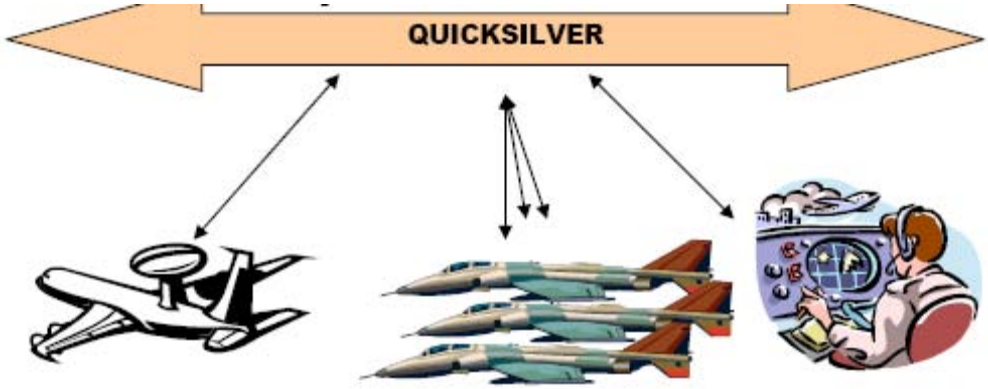

Figure 3: Quicksilver offers a high-throughput message bus scalable to massive data rates and massive numbers of clients and topics.

In the remainder of this section of our final report, we describe our status as of the end of the SRS program with respect to these goals. It should perhaps be emphasized that our work is ongoing despite the fact that we've completed a substantial amount of software and will soon have users. But these are very ambitious projects and we believe that an addition year to eighteen months of work will be needed to complete them. For example, Ricochet and Tempest already comprise some 15,000 lines of code, and Quicksilver is approaching 120,000 lines of code. Implementation, debugging, performance tuning and

associated infrastructure management are hard problems in systems of this size even if one starts with complete knowledge of how they will be developed. In our cases, though, the issues are deeper. In effect, DARPA SRS funding has permitted us to take the critical first steps on what we see as a long term path to develop and provide solutions in these areas. AFOSR, AFRL, NSF and other organizations are now stepping in to pick up funding and hence will enable us to continue the work. Thus, while this is a final DARPA SRS report, it is something of an interim progress report with respect to our broader goals.

At a fundamental level, when we talk about systems that need to guarantee behavior on a scale of thousands or even millions of users, while repelling attacks and overload, and while automatically detecting and repairing faults, we face a new domain of theoretical and scientific questions, and some tough challenges even in measuring the properties of our solutions and comparing them with prior work. To build these systems, we need to advance our understanding of the roles that consistency plays in large-scale systems and understand the tradeoffs between various forms of consistency and the performance implications of offering those guarantees. Service oriented systems are often componentized, and we need to understand how component systems can be composed so as to preserve underlying QoS properties. Thus, to build these systems, we've needed to develop a new science – a science of robust communication in large-scale settings. This science takes the form of a mathematical framework based on the theory of epidemic systems, a new software architecture that focuses on scalability issues, and an experimental strategy for validation of our work in a convincing manner.

As mentioned, metrics have been an ongoing challenge for us. While it is easy to say that "our solutions scale better than prior technologies", demonstrating this is quite another matter. There are no widely accepted standards for evaluating scalability, security or fault-tolerance, and systems for the technology space of interest to us differ in enough ways that any comparison has some degree of apples-to-oranges issues that must be confronted. Yet the development of metrics and comparison with appropriate baseline systems has been a requirement of the SRS program, and we've taken this to heart in our work.

Early in the SRS program, we proposed what we believe are appropriate benchmarks for comparison, particular with respect to Ricochet. The challenge here is that existing real-time communication architectures have been surprisingly haphazard about scalability. The well known DDS (Data Dissemination Service) SOA standard, for example, doesn't scale at all – to send a message, a publisher must have a TCP connection to each destination with which it communicates, and if it employs multiple communication topics, it must have one connection per destination, per topic. Thus, if a publisher is publishing on 10 topics with 50 subscribers each, it would need 500 TCP connections. Systems like Linux and Windows don't allow individual processes to create such large numbers of connections and even with modest numbers, contention effects kill any sort of real-time properties. Thus, we found that comparison with the most obvious standard would suffer from the apples-to-oranges comparison problem mentioned earlier, and hence be inappropriate.

Ricochet does focus on time-critical delivery, but it can also be characterized as a multicast technology for high data rates with extremely strong probabilistic delivery guarantees. In this space, we identified a best of breed technology called Scalable Reliable Multicast (SRM), developed approximately a decade ago under DARPA funding at Berkeley, and still widely popular. SRM turns out to have the identical reliability objectives as Ricochet and the documentation and published papers for the system emphasize the importance of rapid delivery. In fact, SRM was developed (and is still used) in support of internet conferencing systems where low latency is a requirement. Thus, comparison here is more appropriate. Our evaluation focused on the standard metrics for multicast communication: latency until delivery occurs, percentage of packets that needed to be recovered due to some form of loss, percentage that were actually recovered, and overheads associated with the protocol. We evaluated these as we scaled in the various dimensions relevant to the problem domain: numbers of processes, numbers of nodes, rates of failure or message loss, and so forth. Our red team evaluation sought to disrupt the system relative to its goals in these respects. The results were interesting and led to useful insights into ways of improving our system.

In what follows, we give an executive overview of our accomplishments and the outcome of our evaluation studies. Separate papers provide a great deal of detail on Ricochet, Quicksilver, the red-team evaluation, the work we are now doing on Tempest, and so forth; we will not repeat that content here, because those papers have been provided to DARPA as part of our deliverables.

In summary, accomplishments during the SRS program were as follows:

- We developed the underlying scientific principles needed to enable these new platforms. In particular, we showed that existing solutions scale poorly because of questions that come down to a form of “complexity” issue, similar to the algorithmic complexity problems that one faces in building traditional software systems. We demonstrated that by mixing a new generation of gossip (epidemic) protocols with traditional group communication protocols we can overcome the scalability limit of prior solutions. In Ricochet this takes the form of an innovative new use of forward error correction (FEC); Quicksilver offers a powerful and flexible architecture based on introducing one level of indirection by mapping multicast groups seen by the down to multicast regions in which IP multicast or overlap multicast can be performed.

- We implemented our software in a robust form that will soon be widely distributed under a BSD license, with early adopters now lining up.

- We completed experimental studies in which we compared our solutions with best-of-breed baseline systems. For example, in the case of Ricochet, we showed that neither the military real-time standard for communication (a technology called DDS) nor a best-of-breed scalability solution (the widely popular Scalable Reliable Multicast protocol, SRM) can offer the stability, reliability and rapid

delivery of multicasts achieved by Ricochet. Indeed, Ricochet was two to three orders of magnitude faster than any prior solution.

• We completed a red-team exercise in which our solutions "won" except in scenarios where the red team actually found a way to break the cluster communications hardware while operating within the rules of the game, and in a situation where the red team insisted that we disable one of the Ricochet recovery mechanisms (so-called NAK recovery), but then was able to provoke unrecovered packet loss by creating a situation in which NAK recovery was needed. Normally, NAK recovery is a standard part of Ricochet, so both scenarios come down to technical wins for the red team but neither exposed any weakness of Ricochet.

Finally, our solutions have been identified by the Air Force as especially promising technology options for their in-house efforts to develop next-generation GIG and NCES platform support, and we have submitted a proposal to AFRL/IF to continue work on the two systems with the goal of delivering working technologies that AFRL/IF would deploy in early operational settings.

2.1. Ricochet And Tempest

As noted above, *Ricochet* is a multicast layer in support of event notification with time-critical communication requirements. A primary use will be in support of Tempest, a new system for automating the development of scalable time-critical clustered services (Figure 1, 2).

The system targets applications running on clusters or in data centers and was motivated by military applications such as the tracking of potential threats in a radar system. [BBPP05, BPB05] describes the system in more detail, but the basic idea is as follows. We evaluated a number of SOA approaches to building scalable applications in such settings as military radar control systems (for weapons targeting and for air traffic control), e-commerce data centers where rapid customer response times are key, and e-science applications where clusters are used to control scientific experimentation.

Some systems of this type have been documented in the literature – for example, our team played a direct role in helping to develop the French air traffic control system and we understand precisely how that system was architected. Our collaborators at Raytheon contributed perspective from systems under development for DD(X). Other studies were undertaken through dialog with stakeholders, for example at Yahoo!, Amazon and Google. Yet with the exception of papers written about large-scale stock market systems (notably the Swiss system) and some overview materials discussing the French air traffic control system, we are not aware of any credible published studies on the topic of architecting large-scale systems that confront demanding time-critical loads. In fact we were co-authors on the studies just mentioned, and we know of no similar studies by completely independent groups. Thus, we recognize that the approach to identifying open research problems used in our effort was somewhat anecdotal, but it nonetheless represents the best one can hope for at this time. And, anecdotal or not, this stage of

investigation was invaluable in helping us understand what industry is currently viewing as standard best practice in the area, and hence helped us identify problems that can be justified in terms of real user needs. Our own group may write a survey paper to at least get our own findings into wider distribution.

At any rate, this exploration confirmed that the Web Services architecture has become highly dominant: we learned that at present, time-critical applications are typically built by cloning some sort of Web Service application and replicating updates by sending them to the clones, a problem typically referred to as a "multicast", and often implemented as part of a publish-subscribe message bus.

Figure 4 illustrates the idea of a publish-subscribe system used in this manner. We see two groups of clones. In this particular example, the group on the left is sending some form of status update or event notification to the group on the right. For example, the group on the left might have the role of processing raw radar images and the group on the right may be working with radar tracks. An update could represent a change to some track. On the right hand side of the figure we simply explain the terminology of subscribing and publishing, and illustrate the roll of the platform in "managing" the relationships between nodes that publish or subscribe.

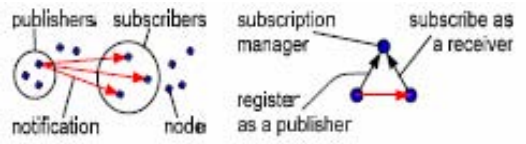

Figure 4. The basic roles: *publishers* and *subscribers* register with the *subscription manager* in order to *send* or *receive* (accordingly) notifications in a given *topic*.

We found that when clusters host large numbers of such applications, a structure emerges in which there are huge numbers of heavily "overlapping" communication groups. For example, if two components of some application are both cloned, each application is likely to have at least one multicast group to send updates to its replicas. If application A sends updates to application B, a group will arise for each category of such updates. Queries are then load-balanced over the replicas, using some appropriate method (in some cases random spraying, in others a scheme more sensitive to affinity or client location). Moreover, data centers or other large-scale structures are often hierarchically organized (for example, a data center might include many clusters of computers). Figure 5 illustrates this perspective and Figure 6 illustrates group overlap.

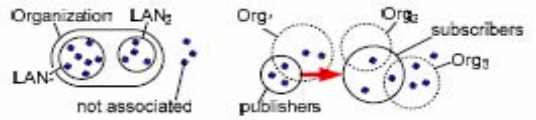

Figure 5. Nodes may belong to *administrative domains* owned by organizations, often divided into a hierarchy of *sub-domains* (e.g. LANs). Publishers/subscribers for a given topic are often scattered across many domains.

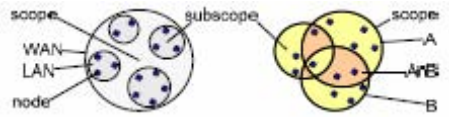

Figure 6. Nodes can also be grouped into a hierarchy of regions based on overlap patterns.

Our premise, as noted above, is that the existing SOA architectures offer inadequate support for time-critical replication, where nodes need to see the updates as rapidly as possible. But the context in which this issue arises is also important, because the setting dictates that time-critical replication will arise in settings characterized by large numbers of highly overlapping groups, that these may have access to hardware support (e.g. hardware multicast), and that they will have a hierarchical administrative structure. Previous work on multicast yielded solution that perform poorly when large numbers of multicast groups are employed and configured with extensive overlap. Thus, the problem as it arises in modern systems points to an area in which new technology development was clearly needed. Moreover, through our dialog with vendors, we found little evidence that the topic will be addressed in future releases of COTS products. In a nutshell, vendors such as IBM and Microsoft showed awareness of the problem, but also felt that the kinds of military systems of concern in our work are a niche area relative to the broader commercial markets on which they are focused. Thus, simply waiting for an off-the-shelf solution would deny the military a technology it needs now.

For example, in a weapons targeting system, updates would correspond to new radar input, and queries might correspond to automated aiming systems that need target location data to aim weapons accurately. The faster the updates reach the application nodes, the better the targeting. But the same system would probably also have other "topics" and hence other, overlapping groups. The processes watching any given radar track would be a group. Processes cooperating on a targeting task might be a group. Thus a radar tracking a dozen possible threats could map to a clustered computing system in which there are literally hundreds of partially or heavily overlapping groups.

Ricochet innovates by using a new kind of forward error correction protocol (FEC) to overcome packet loss, which is an unavoidable problem on modern web service platforms. The scenario to imagine is one in which an urgent update is sent, but dropped in the operating system on some node. That replica will now lag the others until the

update is recovered and delivered. Ricochet works by identifying overlap regions where many groups overlap and then sending a cleverly constructed error-recovery packet that can be used to reconstruct missing data if the amount of missing data is small and the amount of data that got through is large. If a message cannot be recovered because of a correlated loss, the FEC packet still permits the discovery that data is missing. A secondary NAK protocol is then used to recover such a packet from some node with a copy.

The key idea in the FEC construction is to send FEC packets in overlap regions but to use IP multicast to send data packets directly from publishers to receivers. We can "get away" with this in part because of the setting – as explained below, in a LAN or WAN setting, profligate use of IP multicast isn't appropriate. But in a cluster, the match of technology and need turns out to be good. So, Ricochet operates by taking some "window" of data packets in an overlay group, xor-ing them together, and then sending the result in the overlap group as the repair packet. We're currently exploring a stronger coding scheme that might tolerate Byzantine faults, but up to now, have limited the implementation to packet loss or corruption detectable using a checksum.

We have experimented heavily with the protocol and determined that it achieves a three-order of magnitude latency reduction when compared with prior high-speed "scalable" multicast protocols (our comparison was with Scalable Reliable Multicast, a widely popular standard), that it scales far better in the number of groups, and that it also achieves far better real-time delays than the most popular "real-time standard" for SOA settings, a system called the Data Distribution System or DDS.

Ricochet is a complete system and we are now preparing to distribute the software to a number of interested parties, including Raytheon for use in DD(X), Yahoo!, Sun Microsystems, Tangosol, and others. We also plan to use the system in our own follow-on project, as part of Tempest, a new drag-and-drop tool for automating the development of time-critical clustered applications that scale well and self-manage.

3. Quicksilver

As noted above, Quicksilver is a scalable communication system for WAN or LAN settings (Figure 3). It supports a publish-subscribe style of communication in which applications running directly on PCs in a WAN can stream data at high rates to one-another. The focus here is on raw throughput and stability during disruptive episodes, but not on time-critical event delivery.

Quicksilver supports scalability support for hosting time-critical applications in very large-scale settings. Here the intent is to support a scalable publish-subscribe layer for web services applications running on PCs on a massive scale. [OBP05] describes the system in more detail, but the basic idea is as follows. We aim at scenarios involving event notification and publish-subscribe in settings where there are huge numbers of client systems, typically running Windows .NET. Existing publish-subscribe technologies

are relatively Linux and Unix-centric and aimed primarily at networks of workstations of the sort seen in stock trading and other financial settings, which of course are very different from large scale Air Force applications – the networks are of modest size and are very stable by comparison with military networks, failures are rare, loads can be predicted, and the applications running are relatively static.

While the problem may seem very similar to the one we faced in developing Ricochet, it turned out that the setting has a big impact on the appropriate architecture. When products built for stable networks of Linux and Unix workstations (or for clusters) are ported to massive, highly dynamic, unpredictable settings composed of Windows platforms, often mobile and often with disadvantaged communication links (such as will be seen in the Air Force), one finds that they don't scale to an adequate degree and can't guarantee high throughput rates when disruptive events such as failures, configuration changes or load surges occur. Indeed, existing technologies are so easily disrupted that near constant human supervision is often required in large deployments. The same is true throughout the military. Thus, developers face a tough challenge: the GIG/NCES technology decisions will force them to overcome a wide range of readily identified and technically tough challenges.

Thus, our challenge in developing Quicksilver was to break through these scalability and performance barriers while also simplifying the application programming model by embedding scalable communication mechanisms into the runtime environment in ways that are much easier to use than any existing technology. Like Tempest, Quicksilver can support applications written in any of a wide range of programming languages supported by .NET. However, whereas Tempest itself will be coded in Java and will use Linux as a primary platform, Quicksilver is coded in C# and is a native .NET application. Of course, each of these technologies could later be ported into other settings, and because C# is nearly identical to Java, a Java version of Quicksilver would be possible if that became desirable down the road.

The basic idea starts with a similar observation to the one made in Ricochet: publish-subscribe will give rise to huge numbers of heavily overlapping communication groups, offering opportunities to optimize across groups. Here, the main source of overlap is the use of "topics". For example, imagine a publish-subscribe system used in an intelligence gathering setting. A given analyst might be monitoring a large numbers of topics corresponding to a wide range of intelligence subjects, with messages carrying updates on that topic.

Thus we often find group communication patterns in support of what may seem to be non-group patterns. And scalability of group communication is ultimately paramount.

In a nutshell, most existing systems (we have in mind systems such as our own Isis, Horus and Ensemble systems, but also DDS, JGroups, Spread, Transis, Totem, Eternal, SRM, RMTP, etc.) are implemented using a fairly standard "design paradigm" in which one focuses on the use of a single group by a single application (perhaps supporting a few side by side groups that run independently but concurrently in the application's address

space). A sender transmits multicasts using either a series of point-to-point sends (perhaps over TCP channels, in the manner of the DDS real-time message bus, or perhaps with UDP messages or even UDP multicast). Receivers run some form of ACK/NAK protocol, and this provides rate and data loss feedback to the sender, which can retransmit lost data as appropriate. A review of these protocols and a discussion of their limitations can be found in [BIR05].

The systems listed above offer a range of communication guarantees (some of them, like Horus and Ensemble, are actually reconfigurable on the fly to support communication properties that can be dynamically adapted to match the specific needs of a given application [LKR99]). What makes them similar and hence comparable is that they all support some form of groups that can be joined, left (perhaps because of failure or perhaps because the application loses interest in the data associated with the group), and they compete primarily at the level of sustainable data rates. Papers on any of these systems describe experiments in which achieving some sort of all-time record for absolute numbers of multicasts pushed through the system within a single group is clearly the primary goal.

Cornell has held the performance records for nearly a decade [RMB906]. Yet we would be the first to concede that this entire *class* of solutions scales poorly in some respects [BIR05, BIR99, BCH00], and this is the fundamental limitation to which we alluded below. The problems are artifacts of a standard style of implementation. First, if applications participate in huge numbers of multicast groups (for example, think of a publish-subscribe system in which each topic gives rise to a group), there is an explosion of redundant communication and the groups start to contend with one-another. For example, a single application that subscribes to a thousand topics that each have a thousand other subscribers might find itself with millions of communications connections to manage (say, in DDS, where each group needs a connection between each sender and each receiver). A single join or failure event can trigger expensive reconfiguration protocols in all of these thousands of groups at the same time, for example if they must "flush" incomplete messages when a sender leaves (the problem being that a single failure might translate into thousands of "group departure" events, one per group to which the process belonged). Normal multicast traffic contends for resources, with all of these thousands of concurrently active groups fighting for access to the single shared network controller and for buffering space both in the kernel and in the application space.

A possible exception to the rule arises if we focus on what are called lightweight groups; introduced in the Isis system [RGS96,RGV00], these are employed in the Spread platform [SPR01]], a popular package developed at John Hopkins University. But lightweight groups are really just a trick: a single heavy-weight group is presented to users as if it was many lightweight groups using a filtering scheme: messages in the lightweight groups are mapped to multicasts in the heavyweight group, and then undesired incoming messages are dropped by a receiver that isn't really a member of the lightweight group in question. Spread is most often configured to run on a centralized cluster of servers, and can be visualized as a kind of relay service: to send a message, a client sends it down a TCP connection to a Spread agent, which multicasts within a small

heavyweight group containing the other agents (rarely more than four or five), then copies are sent back up to receivers after filtering, again on TCP connections. This results in high latency, and in a large system the servers quickly get overloaded.

We've described one sense in which scalability has been a problem: in the numbers of potentially overlapping groups. But the systems cited above also scale poorly in numbers of users. If a single group has even moderately large size (say, more than 50 members), all sorts of problems are observed. Huge groups typically require some form of hierarchical structure, but this introduces complexity and creates a substantial risk that anomalies will be noticed if a failure disturbs the structure. In SRM [SRM97] (Scalable Reliable Multicast, but perhaps a bit optimistically named!), such events cause overhead to soar quadratically in the size of the group; when a system gets large enough, even modest disruptions will cause storms of overhead and retransmission that can shut the group down; SRM, thus, is an example of a protocol that can only scale in settings that are extremely tranquil, experiencing extremely low rates of packet loss or other changes. Systems like our own Isis, Horus and Ensemble platforms start to exhibit spontaneous logical partitioning problems when more than 100 or so users employ them simultaneously. And while individual protocols, such as our own Bimodal Multicast [BHO99], can scale to much larger settings, they adopt reliability models remote from those one would use to manage security keys or replicated data with strong consistency properties.

In summary, most forms of reliability and QoS ultimately demand scalability of several kinds from some form of group communication infrastructure. Unfortunately, fifteen years of research has yielded mostly systems that support very high data rates in a single group, but only allow applications to join a few groups at a time, and can't handle huge numbers of users.

We mentioned that existing group communication systems also share a more practical limitation that represents a barrier for vendor adoption. This is simply that such systems have never been embedded into standard frameworks in a way that users found easy to use. By and large, group communication systems are just implemented as libraries with their own proprietary APIs and their own non-standard development tools. Some even have non-standard thread libraries. This was a common style of system development in the 1980's and was tolerable then, but today, we need solutions that fit naturally into SOA settings, so that developers can work in standard languages and with standard tools and still exploit those solutions.

Vendors need to see some success stories in both of these respects; lacking them, group communication will remain something of a holy grail for reliability: a tool we know is needed, key to replication, but one that is most often hidden within black boxes like Microsoft's Windows clustering platform or IBM Websphere, and not available for use by general developers. Thus we need to pursue fundamental advances in scalability, and also pursue engineering advances by showing that the solutions can fit naturally into the GIG/NCES platforms that have been identified as standards both for military use and also in the commercial sector. Our belief is that both problems can be solved.

At the core of our Quicksilver system is a technical breakthrough that didn't originate at Cornell or even in the academic community as a whole, but seems to offer a possible response to scalability issues. Some ten years ago, peer-to-peer file sharing emerged within small communities of network users who began to share music and other media (as it turned out, illegally). They attempted to finesse the law by building systems in which there were central lists of potential music sources, but where file sharing was handled directly between the client systems by some form of file "download" occurring directly on the edges of the network. One could use email attachments to solve the download problem, or direct FTP or TCP connections, or any of a number of other schemes, and over time the P2P community began to explore more and more such options. When the first legal cases went badly but seemed focused on the centralized directories, P2P searching and indexing was proposed as a remedy, and one saw a new wave of P2P systems in which everything – file search, download, announcements of new available information – all were handled with P2P protocols; only the initial steps of joining the network used any kind of directory service.

The academic community latched onto the P2P trends around 1997 and a huge wave of research on P2P protocols – not necessarily for illegal content sharing – emerged, with Cornell as one of the leaders. The subsequent period has given us a variety of interesting P2P tools: applications for indexing content on a massive scale and rapidly finding data, new versions of the Internet DNS service that provide rapid update capabilities and faster lookup, P2P multicast protocols (the Bimodal Multicast [BHO99] is an example), P2P software for distributed monitoring, management, data mining and aggregation (our Astrolabe system, now used by Amazon to control their data centers being a good example [RBV03]), etc. The most recent wave of work is giving us P2P file systems, P2P email, P2P chat, P2P gossip overlays, and some widely used products, such as BitTorrent [BTO01], the popular service for distributing software updates and patches.

The experience with these systems has been mixed. P2P solutions have problems with firewalls, since the premise of P2P is that clients should be able to talk directly to other clients, but firewalls and NATs often prevent this. P2P systems are often slow in comparison with server based solutions (because clients are often slower than servers and often have limited connectivity or availability issues), and many kinds of P2P systems are easily disabled by churn (high rates of client arrival/departure/failure), network outages or partitioning failures (these can, for example, leave a P2P index inconsistent and perhaps permanently so), and are also relatively slow to react when configuration changes are required. On the positive side, P2P solutions scale incredibly well, and a subclass of these solutions, in which gossip or "epidemic" communication patterns are employed, are astonishingly robust against the kinds of faults and even DDoS attacks, just enumerated.

By gossip, we refer to a pattern of P2P communication in which nodes periodically (but in an unsynchronized manner) select other nodes pseudo-randomly and exchange state information. Gossip can support epidemic spread of information: first node A knows something. But after A runs a round of gossip, node B will know this information too;

indeed, since A was probably also a gossip recipient, the information may also have reached node C that chose to gossip to A. Thus in one unit of time, we've gone from having just one node that knew the information to three. In two units of time, nine will be "infected", and in general the spread should be exponential, dying out only when all nodes are already infected, which typically occurs within time logarithmic in the size of the network. Because data can travel from node A to node B along what are essentially exponentially many possible paths, there are very few failure patterns that can prevent B from learning whatever A wants to share.

The revolutionary opportunity is simply to combine gossip mechanisms, and similar ideas from the P2P world, with more traditional group communication protocols, so as to benefit from the best aspects of each. For example, we can use traditional high-speed multicast infrastructure to send data, but couple it with a gossip-style of infrastructure to handle recovery of missed packets, tracking of flow control and congestion information, and tracking of membership. Of course this won't solve the problem of allowing a single node to join thousands of groups simultaneously, but we can tackle that by designing smarter data structures that explicitly optimize across sets of groups under the assumption that such patterns will be the common case.

The Quicksilver Reliable Multicast [OBP05] protocol is a highly scalable group communication infrastructure that uses a best-effort reliability model but can scale incredibly well in *all* the dimensions mentioned earlier: numbers of groups, overlap, total numbers of clients in the system, sizes of groups. QSM is setting a new set of performance records: in one configuration, the system seems capable of sending as many as several *million* small updates per second between standard C# applications connected by a standard high-speed bus (the trick is partly one of packing updates into larger messages [FVR97], but even if we disallow this, QSM can send approximately ten thousand 1KB messages per second – an impressive data rate more than an order of magnitude faster than anything ever reported in the past). Here, the thinking is to focus first on extending QSM by embedding it into the .NET framework in a very natural way, by adding group endpoints directly to the .NET common language runtime environment. These can be "typed" endpoints: the type corresponding to the protocol stack (secure, virtually synchronous, best-effort reliable, secure-best-effort, etc). Our plan here is to then layer a pub-sub API over the basic infrastructure within a Web Services eventing model, so that any application built using Web Services can run directly on Quicksilver without modification.

As noted earlier, we believe that Quicksilver will live primarily on client platforms, running primarily Windows operating systems with the Web Services and .NET framework being a dominant model. Thus, Quicksilver itself is coded in C# for .NET and will use Windows as its main runtime environment. Applications can be coded in any of about 25 programming languages ranging from the obvious ones to some very obscure languages, such as OCaml, Visual Basic, and Python.

4. Scalable Byzantine Services

One of the missions of the SRS proposal was to create scalable Byzantine (or Intrusion-Tolerant) services such as aggregation and event notification, while providing high performance. In recent years, Distributed Hash Tables (DHTs) have been proposed to support scalable self-regenerative services. While it is often straightforward to support such services on DHTs, this choice must be seriously questioned, as DHTs dictate routes that are not optimal, and hard to secure.

Originally we set out to try to fix such structures as DHTs and Astrolabe, but we ran into insurmountable difficulties, as these systems were not designed with Byzantine behavior in mind. Here we present a new system that we designed, implemented, and deployed, called *Fireflies*. *Fireflies* provide similar services as traditional DHTs, but can support robust routing between correct participants in the face of (unidentified and possibly undetectable) Byzantine participants.

4.1. Fireflies Structure

One important function of *Fireflies* is to keep track of which participants are stopped, in order to create the largest ratio of correct to Byzantine participants possible. For this, *Fireflies* uses a traditional pinging protocol. A tricky detail is determining how long to wait before issuing an accusation. Using a static global timeout is not a good choice, as this will not scale well and can cause correct members to accuse other correct members more often than necessary. In particular, the timeout period has to adapt to the message loss characteristics between monitor and monitoree. Also, Byzantine members could potentially prevent detections of stopped members by forging ping responses. Our pinging protocol accurately measures message loss statistics in the face of Byzantine members in order to provide detection with a known probability of mistakes.

It is not scalable for each *Fireflies* member to monitor each other member. The Membership component assigns to each member a list of members it is responsible for monitoring. This assignment cannot be under the control of any particular member, or the system would be prone to Byzantine attacks. Upon detection of a failure, a monitor accuses its monitoree and disseminates the accusation over a highly reliable gossip-based broadcast channel, which is described below. Should a Byzantine falsely accuse a member, then the accused member will also receive the accusation and has an opportunity to rebut the accusation using the same broadcast channel.

Fireflies members keep information about each other member. Memory is cheap, so doing so is not unscalable. A public key identifies each member. The members are then organized on k virtual rings using different hash functions. The choice of k is explained below. Each member is assigned to monitor its successor on each ring, and is only allowed to gossip with its predecessor and successor on each of the rings. Doing so significantly restricts the actions of Byzantine participants, as all traffic is signed and participants can quickly determine if received traffic is from one of the ring neighbors.

The number of rings, k, is critical, however. If there are too few, correct members may end up with only Byzantine neighbors. If there are too many, the system may have too much overhead. Using some graph

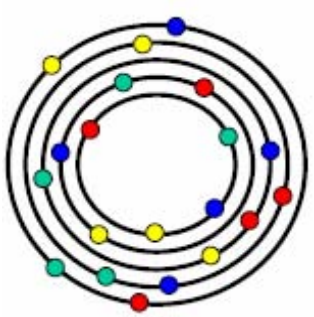

Figure 7: k virtual rings

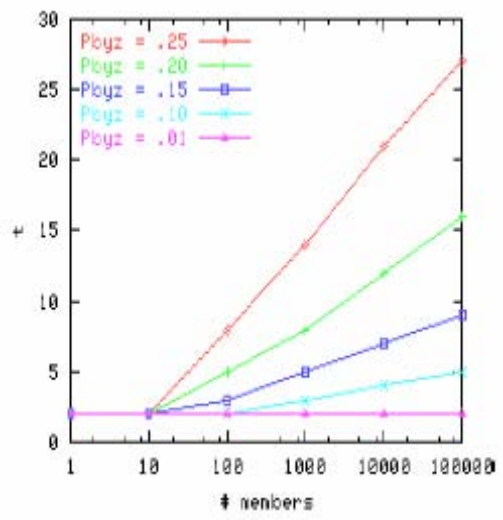

Figure 8: Theoretical Results

theoretical results, we have determined that the number of rings has to grow logarithmically in the number of members. Through Monte Carlo simulations, we can determine the optimal number of rings given the number of members, and the probability that a member is Byzantine, and the desired robustness of the system. The picture to the right shows *t*, where $2t + 1 = k$, as a function of the number of members and the probability of a member being Byzantine.

The graph of correct members and their neighbors, pseudo-randomly determined as above, can be shown to be connected with high probability if k is chosen appropriately. The graph is logarithmic in diameter, and so an epidemic or gossip pair-wise exchange protocol is guaranteed to complete in logarithmic time (logarithmic in the number of

correct members) with high probability. This time can also be determined with high precision using Monte Carlo analysis. Using these parameters, members can determine how long it takes for accusations and rebuttals to disseminate, and use this information to create a correct view of the membership with high probability.

The last part of *Fireflies* is the protocol that does the pair-wise exchanges. The simplest approach would be to have the members exchange their entire state each time they gossip. Doing so would not only be highly inefficient, it would allow Byzantine members to put an unduly high load on correct members. We use various techniques in order to make the exchanges highly efficient. Since the neighbors are determined pseudo-randomly, and there are only O(log N) neighbors, it is possible to have persistent connections between neighbors. We combine this with a set reconciliation protocol that ensures that the amount of information that is exchanged is only slightly larger than the size of the differences in state between the two neighbors. The rate at which information is sent is also limited to further prevent Byzantine attacks on this part of the protocol.

4.2. Fireflies Deployment

Fireflies has been implemented in Python and has been running for over a year on the PlanetLab infrastructure, comprising hundreds of nodes scattered around the world. We have completed many experiments on this deployment, one of which we will present below.

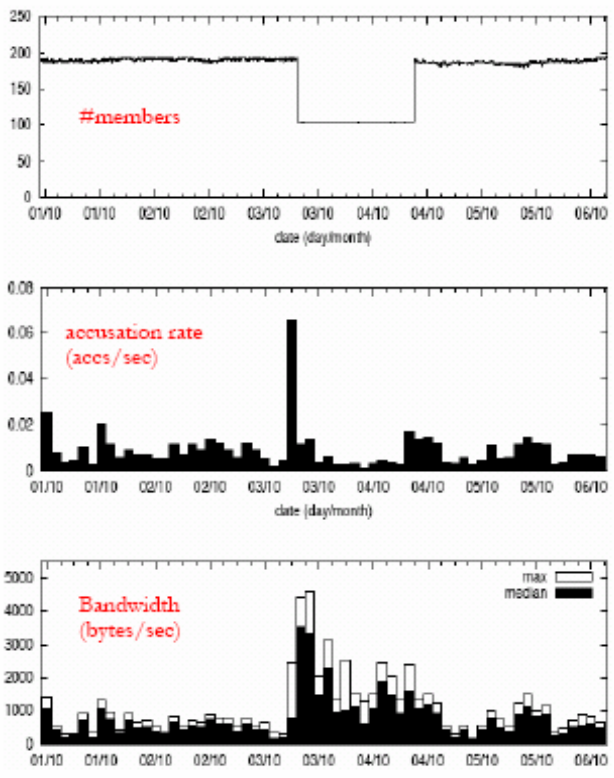

Figure 9: Fireflies Experimental Data

In this experiment, we ran the system on close to 200 nodes in PlanetLab. The experiment ran between October 1st and October 6th, 2005. On October 3rd, we killed the *Fireflies* agents on half of the nodes, chosen randomly. We recovered them a day later. The top graph shows the number of members. The second graph shows the rate of accusations made on the broadcast channel. The spike coincides with the killing of the agents.

More importantly, the bottom graph shows the amount of bandwidth consumed. While low, one can see that there is increased communication throughout the period that the nodes are down. The reason is that there is still some churn (nodes coming and going) going on in the background, and the outstanding accusations create a continuous extra load.

4.3. Fireflies Ongoing Work

A paper on *Fireflies* was accepted to Eurosys 2006, detailing the protocol and experiment evaluation [vRJ06]. But work on Fireflies is ongoing. On the implementation side, we are improving set reconciliation in order to bring down the consumed network resources further. On the verification side, we have nearly completed a correctness proof. But most work is on the development of applications.

We have implemented an intrusion-tolerant shared memory facility, which is the first step towards an information management facility like Astrolabe. As in Astrolabe, each member has a MIB containing attributes about that member. These MIBs can be updated by the member only, and are reliably disseminated to all other members. One can think of this as a single-domain version of Astrolabe, albeit intrusion-tolerant. Updates are flooded rather than gossiped, and thus dissemination is significantly faster than in Astrolabe, with a likewise improvement in freshness of information. In order to increase scale, the next step is to add an aggregation facility. We have done some preliminary theoretical work on Byzantine aggregation, but have yet to finish this work and start an implementation.

We have also implemented an intrusion-tolerant multicast facility, based on the ChainSaw protocol [PKT05]. ChainSaw is not intrusion-tolerant, but with slight modification ChainSaw can be run on *Fireflies* and an efficient Byzantine multicast facility results. We are currently looking at various aspects of this protocol. All traffic is signed; should one be pessimistic and check signatures before forwarding, or gain significant improvement in delays by first forwarding a message and then checking the signature? Also, are all neighbors equals, or should we prefer some neighbors to others? A tit-for-tat strategy can prevent freeloading, but it can also result in bad throughput. These and other issues are currently under consideration, and will soon result in another paper.

Finally, we are in the process of designing a full-blown news dissemination service, or a pub/sub service if you will, in collaboration with colleagues at the University of Tromsø in Norway. This will encompass both the shared memory and multicast facilities, but enhanced with sophisticated subscription interfaces.

5. Cayuga: Stateful Publish/Subscribe

Publish/Subscribe is a popular paradigm for users to express their interests ("subscriptions") in certain kinds of events ("publications"). Traditional publish/subscribe (pub/sub) systems such as topic based and content based pub/sub systems allow users to express stateless subscriptions that are evaluated over each event that arrives at the system; and there has been much work on efficient implementations [1]. However, many applications require the ability to handle stateful subscriptions that involve more than a single event, and users want to be notified with customized witness events as soon as one of their stateful subscriptions is satisfied.

Traditional pub/sub systems scale to millions of registered subscriptions and very high event rates, but have limited expressive power. In these systems, users can only submit subscriptions that are predicates to be evaluated on single events. Any operation across multiple events must be handled externally. In Cayuga, however, subscriptions can span multiple events, involving parameterization and aggregation, while maintaining scalability in the number of subscriptions and event rate. In comparison, full-fledged Data Stream Management Systems (DSMS) [2, 3, 4] have powerful query languages that allow them to express much more powerful subscriptions than stateful pub/sub systems; however, this limits their scalability with the number of subscriptions, and existing DSMSs only do limited query optimization. Figure 10 illustrates these tradeoffs.

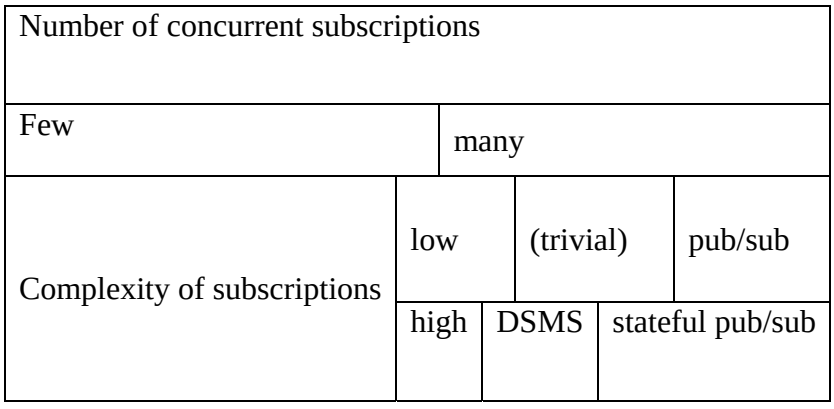

Figure 10: Tradeoffs Between Complexity of Subscriptions and Scale

Another area very closely related to stateful pub/sub is work on event systems. Event systems can be programmed in languages (called event algebras) that can compose complex events from either basic or complex events arriving online. However, we have observed an unfortunate dichotomy between theoretical and systems-oriented approaches in this area. Theoretical approaches, based on formal languages and well-defined semantics, generally lack efficient, scalable implementations. Systems aproaches usually lack a precise formal specification, limiting the opportunities for query optimization and query rewrites. Indeed, previous work has shown that the lack of clean operator semantics can lead to unexpected and undesirable behavior of complex algebra expressions [5]. Our approach was informed by this dichotomy, and we have taken great care to define a language that can express very powerful subscriptions, has a precise formal semantics, and can be implemented efficiently.

Cayuga is a stateful publish/subscribe system based on a nondeterministic finite state automata (NFA) model. In this report, we introduce the Cayuga event algebra and the associated automata model. We also overview the implementation of our system which leverages techniques from traditional pub/sub systems as well as novel MultiQuery Optimization (MQO) techniques to achieve scalability

5.1. Data Model

Our event algebra consists of a data model for event streams plus operators for producing new events from existing events. An event stream, denoted as S or Si, is a (possibly infinite) set of event tuples (a,t0,t1). As in the relational model, a=(a1,...,an) are data values with corresponding attributes (symbolic names). The ti's are temporal values representing the start (t0) and end timestamps (t1) of the event. We assume each event stream has a fixed schema, and events arrive in temporal order. That is, event e1 is processed before e2 iff e1.t1 <= t2.t1. However, a stream may contain events with non-zero duration, overlapping events and simultaneous events (events with identical time stamp values). Our operator definitions depend on the timestamp values, so we do not allow users to query or modify them directly. However, we do allow constraints on the duration of an event, defined as t1 – t0 + 1(we treat time as discrete, so the duration of an event is the number of clock ticks it spans). We store starting as well as ending timestamps and use interval-based semantics to avoid well-known problems involving concatenation of complex events [5].

5.2. Operators

Our algebra has four unary and three binary operators. We give here only a brief description of them here; a formal definition and more examples can be found in our published work resulting from this project [6].

The first three unary operators, the projection operator πX, the selection operator $\sigma\theta$, and the renaming operator ρf are well known from relational algebra. Projection and renaming can only affect data values; temporal values are always preserved. As the renaming operator only affects the schema of a stream and not its contents, we will often

ignore this operator for ease of exposition. Instead, we will denote attributes of an event using the input stream and a dot notation, making renaming implicit. For example, the name attribute of events from stream S1 will be referred to as S1.name. A selection formula is any boolean combination of atomic predicates of the form $\tau 1$ relop $\tau 2$, where the τi are arithmetic combinations of attributes and constants, and relop can be one of <, =, >, or string matching. We also allow predicates of the form DUR relop c where the special attribute DUR denotes event duration and c is a constant. The unary operators above enable filtering of single events and attributes, equivalent to a classical pub/sub system. Subscription S1 is an example of such a stateless subscription.

The added expressive power of our algebra lies in the binary operators, which support subscriptions over multiple events. All of these operators are motivated by a corresponding operator in regular expressions. The first binary operator is the standard union operator. Our second operator is the conditional sequence operator S1;θ S2. For streams S1 and S2, and selection formula θ (a predicate), S1;θ S2 computes sequences of two consecutive and non-overlapping events, filtering out those events from S2 that do not satisfy θ. Adding this feature is essential for parameterization, because θ can refer to attributes of both S1 and S2. This enables us to express "groupby" operations, i.e., S1;θ S2 essentially works as a join, combining each event in S1 with the event immediately after it in S2. However, θ works as a filter, removing uninteresting intervening events. Subscriptions S2 and S3 are examples of such subscriptions. Our third binary operator is the iteration operator $\mu F,\theta$ (S1,S2), motivated by the Kleene+ operator. Informally, we can think of $\mu F,\theta$ (S1,S2) as a repeated application of conditional sequencing: (S1;θ S2) union (S1;θ S2;θ S2) union … Each clause separated by the union operator corresponds to an iteration of processing an event from S2 which satisfies θ. The additional parameter F, a composition of selection, projection and renaming operators, enables us to modify the result of each iteration. Thus μacts as a fixed point operator, applying the operator ;θ on each incoming event repeatedly until it produces an empty result. To avoid unbounded storage, at each iteration, it will only remember the attribute values from stream S1 and the values from the most recent iteration of S2. For any attribute ATTi in S2, we refer to the value from the most recent iteration via ATTi.last. Initially, this value is equivalent to the corresponding attribute in S1, but it will be overwritten by each iteration.

5.3. Mapping to Automata

Given the algebra's similarity to regular expressions, finite automata would appear to be a natural implementation choice. Similar to the classic NFA model, for an incoming event, an automaton instance in one state can explore all the outgoing edges, and nondeterministically traverse any number of them. If it cannot traverse any edge, however, this instance will be dropped.

We extend standard finite automata in two ways. First, attributes of events can have infinite domains, e.g., text attributes, and therefore the input alphabet of our automaton, which is the set of all possible events, can be infinite as well. To handle this case, we associate each automaton edge with a predicate, and for an incoming event, this edge is traversed if the predicate is satisfied by this event. Second, to be able to generate

customized notification and to handle parameterized predicates over infinite domains, we need to store in each automaton instance the attributes and values of those events that have contributed to the state transition of this instance. These attributes and values are called bindings. To avoid overwriting the bindings of earlier events with that of latter events, we also need an attribute renaming function for each edge so that when an event makes an automaton instance traverse that edge, the bindings in that event are properly renamed before being stored in the instance.

We have developed a mechanical way to translate algebra expressions into automata. Details of this mechanism as well as the proof of correctness can be found in our technical report [7]. Intuitively, for a given algebra expression, we first construct a parse tree, and then translate each tree node corresponding to a binary operator into an automaton node. In our mechanism any leftdeep parse tree can be translated into a single automaton, referred to as a leftdeep automaton.

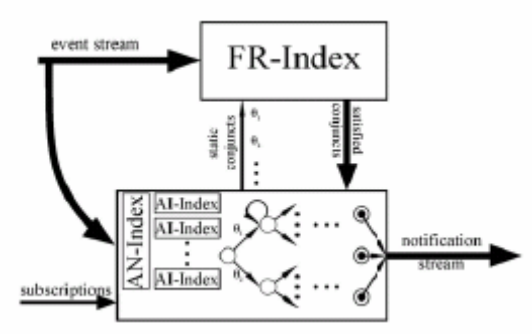

Figure 11: Cayuga Architecture

5.4. Architecture

The overall system architecture of Cayuga is shown in Figure 11. Its core component is the State Machine Manager, which manages the merged query structures and the automaton instances at the nodes. It also maintains two auxiliary index structures, the ANIndex and the AIIndex. Outside the State Machine Manager, there is a third auxiliary index structure, the FRIndex.

5.5. Performance

We illustrate the performance of Cayuga through a Cayuga application scenario that was developed by Raytheon, our partner in this DARPA program (and parts of the following paragraph are taken verbatim from their document describing various application scenarios). Consider that war fighters are outfitted with a Personal Digital Assistant (PDA) that contains a Global Positioning System (GPS). The PDA is wirelessly connected to a military information system so that up-to-date positioning information is

transmitted. In addition, a database stores dangerous areas in the Iraqi theatre that are considered hazardous. Perhaps they are the locations were there is a high incidence of Improvised Explosive Devices (IEDs), or areas like Fullujah that have been under the control of insurgents.

In the Iraqi theatre, nearly 30% of the fatalities are caused by Improvised Explosive Devices (IEDs). It could be determined that in some areas, the probability of an IED explosion is much higher wherever an insurgent has been observed. This could result in the following query:

Example Query. Notify any troop located in Iraq who is within 5 km of any location where an insurgent has been observed within the past 24 hours.

Note that the system has to remember where every insurgent had been over the past 24 hours. Query 4 also demonstrates that as the qualitative query becomes more multifaceted, the magnitude of the processing intensity grows. As more systems are involved, the sophistication and distributed nature of the subscriptions become more complex and the number of events is increased.

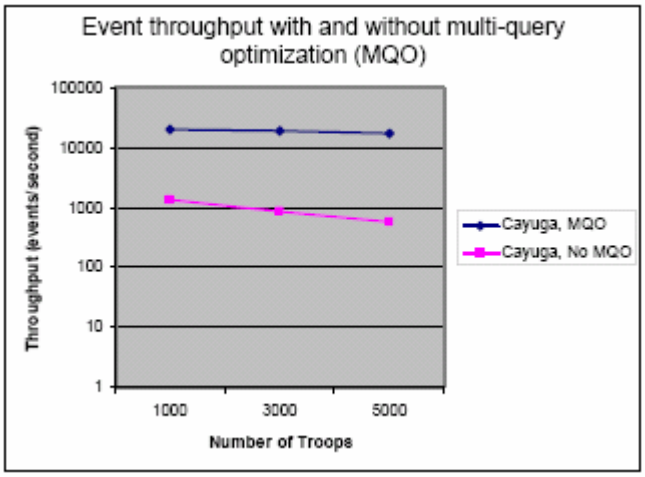

Figure 12: Cayuga Event Throughput

Raytheon provided the following parameters for data generation. The number of troops is varied from 1000 to 5000. Each troop emits 1 event/minute to report their location. An insurgent event is created every hour. There are 10 troop areas, Iraqis are in one area. Troops and insurgents are uniformly distributed. Figure 9 shows the performance of Cayuga with and without its performance optimizations (MQO). The performance of Cayuga without MQO would be the performance of a traditional publish-subscribe system with extensions for stateful queries through a middleware layer. The figure shows that Cayuga provides nearly two orders of magnitude performance improvements over existing systems.

6. Chunkyspread

The Chunkyspread [VEN06] component of Quicksilver is capable of delivering a real-time, high-volume content stream to tens of thousands of recipients. Chunkyspread is a "P2P" system in that recipients of the stream (members) are also responsible for delivering the stream to other recipients. A key aspect of Chunkyspread, however, is *load heterogeneity* members can transmit as much or as little as they wish. This means that high-capacity infrastructure nodes can contribute to stream delivery as appropriate. In this sense, Chunkyspread can be pure P2P, pure infrastructure-based, or some combination of the two. We see this kind of flexibility as being central to highly dynamic deployment scenarios. Imagine, for instance, a video streaming session originating at a central facility and being distributed to a set of mobile devices using only the resources of that equipment. Suddenly those devices must be used for an additional purpose, such as chatter among the users of the devices during an engagement, and the mobile devices no longer have the capacity to both distribute and receive the stream. Other equipment could then be dynamically exploited to handle the stream distribution.

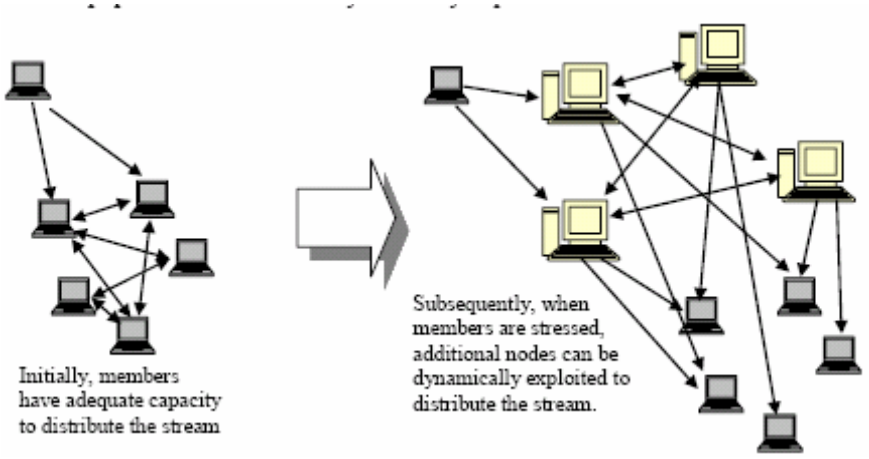

Figure 13: Initial and Subsequent Results

At the beginning of the SRS contract period, the baseline technology for overlay multicast streaming was Splitstream [CDK03]. Splitstream is research, not commercial, technology. At the time, there existed no appropriate commercial baseline for large-scale streaming. This is because commercial real-time streaming deployments such as Akamai [AKA06] or Real Networks [REA06] are purely static infrastructure deployments. They are not self-configuring or self-healing in any way. Since the start of the SRS contract period, however, a number of Chinese startup companies, for instance [PPL06], have been offering P2P real-time streaming for IPTV (television over IP) at a scale of a few tens of thousands of peers. The underlying multicast is coupled to the IPTV application, and is therefore not available as a middleware component. Furthermore, the underlying multicast technology is proprietary. During a recent trip to China, however, we were able to determine that the technology is similar to a research approach called Chainsaw

[PKT05]. Chainsaw is a real-time variant of BitTorrent [COH03]. BitTorrent is the latest and hottest file-sharing technology.

We were able to go beyond our originally stated goal of using Splitstream as our sole baseline, and here present baseline comparisons for both Splitstream and Chainsaw. The criteria of interest are:

1. Load heterogeneity: Members must have fine-grained control over their load. This leads to the deployment flexibility discussed above. It also leads to improved system throughput, because control over load allows each member to best utilize its' transmit capacity and therefore maximize throughput.

2. Throughput: Achieving high throughput is closely related to control over load.

3. Latency: Transit time from source to all receivers should be minimal. While this goes without saying, we should stress that minimizing latency is quite difficult in the context of a P2P style of distribution. While hardware or IP multicast can achieve lower latencies than is possible with a P2P approach, IP multicast tends to be available only in local environments.

4. Convergence time: We want to minimize both the time it takes for a member to start receiving the stream, and the time it takes for a member to reach its target load.

5. DoS attack resistant: Members should not be able to force other members to waste transmit capacity. In particular, a tit-for-tat approach can prevent this.

To understand the differences in the performance of the three approaches, we need to first understand roughly how they operate. The cited papers provide more detail. All three approaches have certain high-level characteristics in common. That is, a node wishing to receive the multicast stream first discovers some number of already joined members. The joining node then selects a subset of these discovered nodes and starts receiving different portions of the stream from different selected nodes. Once the joining nodes starts receiving the stream, then it starts transmitting portions of the stream to other nodes. This must be done is such a way that there are no loops in the different paths taken by different portions of the stream.

Splitstream takes a structured approach to establishing these paths. Nodes start by joining a Distributed Hash Table (DHT) called Pastry [ROW01]. DHTs in general, and Pastry in particular, have some specific internal structure that allows messages to be routed between any pair of nodes in under log(N) hops, where N is the number of member nodes. Splitstream exploits this structure to form multiple loop-free trees along which different slices of the stream may travel. A slice is a recurring periodic packet in the stream. For instance, if there are 16 slices, then the 1^{st} packet belongs to the 1^{st} slice, the 2^{nd} packet to the second slice, and so on, with the 16^{th} packet belonging again to the 1^{st} slice, the 17^{th} packet to the 2^{nd} slice, and so on.

Structured		Unstructured
Per-slice Trees	Splitstream	Chunkyspread
No Trees		Chainsaw

Figure 14 Comparison – Unstructured, Structured

One of our original goals for Chunkyspread was to avoid the structured approach. We were motivated by the fact that almost all successful P2P applications to date, for instance Gnutella, Kazaa, and BitTorrent, are unstructured. In such an approach, nodes randomly select other nodes with which to transmit and receive portions of the stream. Such an unstructured approach better reflects natural systems, results in simpler algorithms, and as such will be more robust. Note, however, that it is hard to measure this robustness. This is in part because simpler algorithms tend to be more robust because they have fewer bugs. In a research lab setting, however, one naturally removes all the bugs one discovers and so the advantage of simplicity tends to disappear. It would be in operational settings when (unknown) bugs would finally manifest themselves. As such, while we are not in a position to provide good metrics to demonstrate the fundamental robustness advantage of unstructured over structured approaches, we must never-the-less stress that this advantage is terribly important.

In Chunkyspread, each node scalably discovers randomly selected nodes using random walks. We use an approach developed by us specifically for Chunkyspread called Swaplinks [VIS06]. Swaplinks has the unique property that each node can statistically control the number of neighbors that it discovers, and that discover it. As a result, nodes that wish to have a higher transmit load discover proportionally more random neighbors. Once discovered, each node enters into local negotiations with its neighbors to determine which slices of the stream it will exchange with each neighbor. These negotiations take various criteria into account. One such criterion is of course loop-freeness—as with Splitstream, each slice travels along a tree. Chunkyspread uses bloom filters, carried in all stream packets, to avoid and discover loops. Otherwise, our criteria include load, latency, and tit-for-tat, reflecting the three of the primary metrics in our evaluation.

Chainsaw, as well as its commercial Chinese counterparts, takes the unstructured approach to an even further extreme. In addition to selecting random neighbors, Chainsaw dispenses with trees altogether. Their idea is that even the formation of trees represents too much structure. Instead, Chainsaw breaks the stream into blocks, where a block consists of some number of contiguous packets. During transmission, each node advertises to its neighbors which blocks it has received, and explicitly requests from its neighbors the blocks that it does not yet have. This approach is patterned after the popular BitTorrent file-sharing application—it is essentially a real-time BitTorrent.

With this background in place, we can now address our performance metrics. Figure 15 is a CDF showing how well nodes in both Splitstream and Chunkyspread can control their load. Note that Chainsaw is not included in this graph because Chainsaw was not designed for load heterogeneity. Even with modifications, it is not at all clear whether it

would be possible to get some kind of load heterogeneity out of Chainsaw. The graph of Figure 15 is for a simulated 5000-member network, where 3750 members are in place (have completed the neighbor discovery phase) at the beginning of the stream, and the remaining 1250 members join from the 20th second of the simulation at a rate of 50 joins per second. The target load of each member varies randomly within a ratio of 5:1. The total capacity of members is adequate to distribute the stream to all members.
Figure 15 shows that Chunkyspread achieves far better control over load than does Splitstream. The vast majority of Chunkyspread nodes achieve within 20% of their load target. In the scenario where Chunkyspread ignores the latency criteria, 90% of members are within 10% of their load target. By contrast, with Splitstream only 15% of the members are within 10% of their target load. Roughly 30% of the members are loaded to their maximum capacity, while 25% of the members operate at less than 50% of their capacity. Nearly 5% of the members transmit nothing whatsoever.

Figure 15: Control over transmit load.

The take-away from Figure 15 is that Chunkyspread is the only existing overlay multicast technology that utilizes node capacities well. While all Chunkyspread members are operating comfortably within their capacity margins, many Splitstream members are completely maxed out. Operating systems at maximum capacity is a bad idea—it leads to increased failure modes and unpredictable behavior. One way to avoid this, of course, is for members to select a configured maximum load that is well within their actual maximum load. This effectively moves the Splitstream curve in Figure 15 to the left, which ultimately leads to reduced overall system throughput because of the members that run under-capacity.

Regarding latency, Chunkyspread and Splitstream perform similarly, with each exhibiting a latency of roughly five times that achievable by IP multicast. In absolute numbers, this means that if the average network latency is 30 ms (a typical number for wide-area Internet paths), Chunkyspread and Splitstream will see average latencies of roughly 150ms. By contrast, the published Chainsaw experiments show latencies of between 1.5 and 2 seconds—roughly ten times worse. The reason for this is the time it

takes for each node to notify its neighbors as to what blocks of data it has received, for the neighbors to react and request certain blocks, and for those blocks to be delivered. If Chainsaw wishes to reduce the overhead of these report and request messages (currently a report is sent for every block), then it must delay the sending of the notifications so as to report multiple blocks in one message, which further increases the stream latency.

Tree-based approaches don't experience this added delay. The take-away here is that Chunkyspread hits the simplicity sweet-spot. Splitstream is far more complex without providing any performance benefit what-so-ever (and performs worse by some measures). Chainsaw is arguably simpler, but pays for this with either high packet latency or high control message overhead.

Convergence times for all three approaches are similar and quite respectable. In our simulations, joining Chunkyspread member start receiving the full on average within 5 seconds, and no joining member took longer than 10 seconds to start receiving the stream.

A crucial aspect of any P2P system is the ability to prevent misbehaving nodes from disrupting the system. While in consumer systems the main concern is with freeloaders, military systems have to contend with active attackers. A particularly nasty form of attack is where the attacker fools many systems into transmitting large volumes of traffic to the attacker (in lieu of transmitting traffic to legitimate members). The volume may be many times the send or receive capacity of the attacker. This both wastes the senders' resources and causes network congestion in the vicinity of the attacker. An effective deterrent to this attack is to require tit-for-tat exchange of data packets. This limits the impact of the attacker by the amount of transmit resources available to the attacker. While Chunkyspread members could not impose perfect 1:1 tit-for-tat on their neighbors, a ratio of 3:4, whereby a node will send 4 slices to a neighbor so long as the neighbor sends 3 in return, was achievable. Doing so resulted in a latency penalty of roughly 50%, but did not impact control over load.

Note that no other real-time multicast protocol, including Splitstream or Chainsaw, has tit-for-tat capability. We don't believe that it would be possible to add tit-for-tat to Splitstream. Indeed the title of the Splitstream publication (Splitstream: High-Bandwidth Multicast in *Cooperative* Environments) indicates that this is the case. While it might be possible to add tit-for-tat to Chainsaw (and other BitTorrent-like approaches), we are doubtful that this would work well. The reason is because finding the appropriate set of parent-child relationships requires some searching. In Chunkyspread this search is amortized over the lifetime of the neighbor relationship. With Chainsaw, the search must begin anew with each block of data, and might easily be cost prohibitive.

6.1. Chunkyspread Conclusions

At the beginning of the SRS project, we believed that Chunkyspread would significantly out-perform existing state-of-the-art P2P multicast protocols by important metrics of load heterogeneity, throughput, latency, and response time. While this turns out to be true for the first three of these metrics, we now believe that a stronger statement can be made. Namely, that Chunkyspread is the only existing P2P multicast protocol that is suitable for a broad range of military applications. The main reason for this is Chunkyspread's unique amenability to tit-for-tat operation, combined with the fact that Chunkyspread performs as well or better than its competitors on all metrics.

7. Complete List of Publications & References

1. [MBvR05] A Scalable Services Architecture. Tudor Marian, Ken Birman, and Robbert van Renesse. In submission

2. [OBP05] The Power of Indirection: Achieving Multicast Scalability by Mapping Groups to Regional Underlays. Krzysztof Ostrowski, Ken Birman, and Amar Phanishayee. In submission.

3. [BBPP05] Ricochet: Low-Latency Multicast for Scalable Time-Critical Services. Mahesh Balakrishnan, Ken Birman, Amar Phanishayee, and Stefan Pleisch. In Submission. 2005

4. [Bir05b] Can Web Services Scale Up? Ken Birman. IEEE Computer. Volume 38. Number 10. Pgs.107-110. October 2005

5. [vRB05] Autonomic Computing - A System-Wide Perspective. Robbert van Renesse and Kenneth P. Birman. "Autonomic Computing: Concepts, Infrastructure, and Applications", ed. Manish Parashar and Salim Hariri, CRC press, 2006.

6. [BPB05] Slingshot: Time-Critical Multicast for Clustered Applications. Mahesh Balakrishnan, Stefan Pleisch, Ken Birman. IEEE Network Computing and Applications 2005 (NCA 05). Boston, MA

7. [Bir05a] Reliable Distributed Systems Technologies, Web Services, and Applications. Birman, Kenneth P. 2005, XXXVI, 668 p. 145 illus., Hardcover ISBN: 0-387-21509-3 Springer Verlag.

8. [BHP05] Building network-centric military applications over service oriented architectures. Kenneth Birman, Robert Hillman, Stefan Pleisch. SPIE Defense and Security Symposium 2005. March 29-31, 2005. Orlando, Florida.

9. [vRJ06] Fireflies: Scalable Support for Intrusion-Tolerant Network Overlays. Robbert van Renesse, Håvard Johansen, and André Allavena. Proceedings of Eurosys, Leuven, Belgium, April 2006.

10. [PKT05] Chainsaw: Eliminating Trees from Overlay Multicast. V. Pai, K. Kumar, K. Tamilmany, V. Sambamurthy, and A.E. Mohr. In Proc. of the 4th Int. Workshop on Peer-to-Peer Systems, Ithaca, NY, February 2005.

11. F. Fabret, H.¬A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In Proc. SIGMOD, pages 115– 126, 2001.

12. D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the borealis stream processing engine. In Proc. CIDR, pages 277–289, 2005.

13. R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In Proc. CIDR, 2003.

14. S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In Proc. CIDR, 2003.

15. D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. In Proc. ICDE, pages 392–399, 1999.

16. Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards Expressive Publish/Subscribe Systems. To appear in Proceedings of the 10th International Conference on Extending Database Technology (EDBT 2006), Munich, Germany, March 2006.

17. A.Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. A general algebra and implementation for monitoring event streams. Technical report, Cornell University, 2005. http://techreports.library.cornell.edu.

18. [AKA06] www.akamai.com

19. [CDK03] Splitstream: High-Bandwidth Multicast in Cooperative Environments. M. Castro, P. Druschel, A. M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. In SOSP, 2003

20. [COH03] Incentives Build Robustness in BitTorrent. B. Cohen. In The First

21. Workshop on Economics of Peer-to-peer Systems, June 2003.

22. [REA06] www.real.com

23. [PKT05] Chainsaw: Eliminating Trees from Overlay Multicast. V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. E. Mohr. In the Fourth International Workshop on Peer-to-Peer Systems IPTPS 2005, Feb. 2005

24. [PPL06] www.pplive.com

25. [ROW01] Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. A. Rowstron and P. Druschel. In Middleware, November 2001

26. [VEN06] Chunkyspread: Multi-tree Unstructured End System Multicast. Vidhya Venkatraman and Paul Francis. In the Fifth International Workshop on Peer-to-Peer Systems IPTPS 2006

27. [VIS06] On Heterogeneous Overlay Construction and Random Node Selection in Unstructured P2P Networks. Vivek Vishnamurthy and Paul Francis. IEEE INFOCOM 2006, April 2006